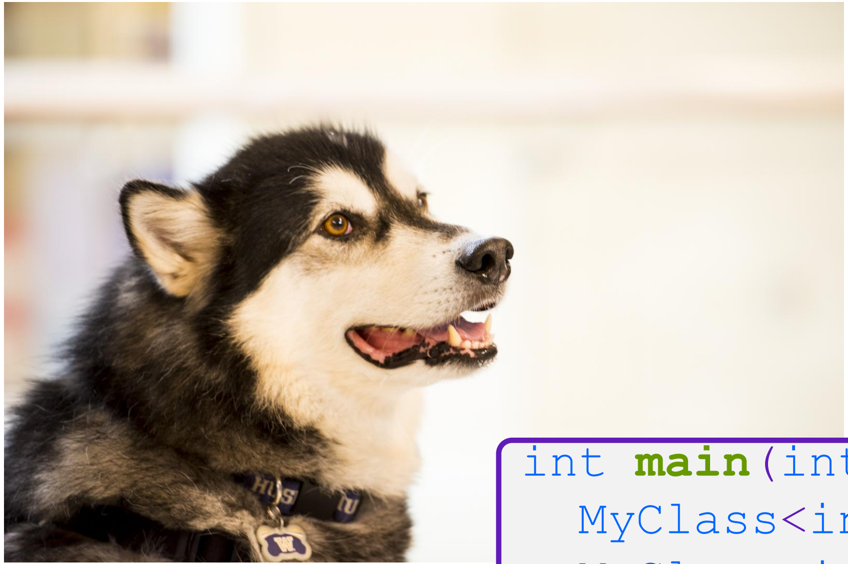
What do you think?

Which lines fail compilation?



- Assume a templated `MyClass` which has an (explicit) single-argument constructor
- `MyClass` has disabled its copy constructor and assignment operator

```
int main(int argc, char **argv) {
  MyClass<int> x(5);   // line 1
  MyClass<int> y(x);   // line 2
  MyClass<int> z;      // line 3
  z = x;               // line 4
  return EXIT_SUCCESS;
}
```

CSE 374: Lecture 26

Smart Pointers

Thanks to Hannah Tang & Alex Mckinney

Pair Class Definition

Pair.h

```
#ifndef PAIR_H_
#define PAIR_H_

template <typename Thing> class Pair {
 public:
  Pair() { };

  Thing get_first() const { return first_; }
  Thing get_second() const { return second_; }
  void  set_first(Thing& copyme);
  void  set_second(Thing& copyme);
  void  Swap();

 private:
  Thing first_, second_;
};

#include "Pair.cc"
#endif  // PAIR_H_
```

Template parameters for class definition

Could be objects, could be primitives

Common C++ STL Containers (and Java equiv)

Sequence containers can be accessed sequentially

- **vector<Item>** uses a dynamically-sized contiguous array (like ArrayList)
- **list<Item>** uses a doubly-linked list (like LinkedList)

Associative containers use search trees and are sorted by keys

- **set<Key>** only stores keys (like TreeSet)
- **map<Key, Value>** stores key-value pair<>'s (like TreeMap)

Unordered associative containers are hashed

- **unordered_map<Key, Value>** (like HashMap)

Smart Pointers

Intro and `toy_ptr`

Smart Pointers 101

C++ Smart Pointers

A smart pointer is an **object** that stores a pointer to a heap-allocated object

- A smart pointer looks and behaves like a regular C++ pointer
  - By overloading `*`, `->`, `[]`, etc.
- These can help you manage memory
  - The smart pointer will delete the pointed-to object *at the right time* including invoking the object's destructor
    - When that is depends on what kind of smart pointer you use
  - With correct use of smart pointers, you no longer have to remember when to `delete` heap memory! (*If* it's owned by a smart pointer)

A Toy Smart Pointer

We can implement a simple one with:

- A constructor that accepts a pointer
- A destructor that frees the pointer
- Overloaded * and -> operators that access the pointer

A smart pointer is just a Template object.

ToyPtr Class Template

ToyPtr.h

```
#ifndef _TOYPTR_H_
#define _TOYPTR_H_

template <typename T> class ToyPtr {
 public:
  ToyPtr(T* ptr) : ptr_(ptr) { }     // constructor
  ~ToyPtr() { delete ptr_; }         // destructor
```

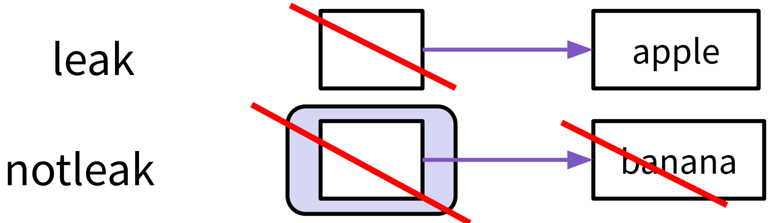
Takes advantage of implicit calling of destructor to clean up for us

```
  T& operator*()  { return *ptr_; }  // * operator
  T* operator->() { return ptr_;  }  // -> operator

 private:
  T* ptr_;                           // the pointer itself
};

#endif  // _TOYPTR_H_
```

ToyPtr Example

usetoy.cc

```
#include <iostream>
#include "ToyPtr.h"

int main(int argc, char **argv) {
  // Create a dumb pointer
  std::string* leak = new std::string("apple");

  // Create a "smart" pointer (OK, it's still pretty dumb)
  ToyPtr<std::string> notleak(new std::string("banana"));

  std::cout << "     *leak: " << *leak << std::endl;
  std::cout << "  *notleak: " << *notleak << std::endl;
  return 0;
}
```

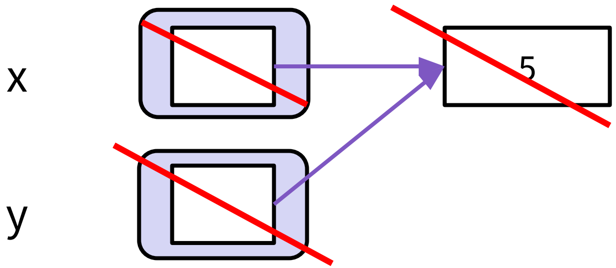


Demo: ToyPtr

ToyPtr Class Template Issues

toyuse.cc

```
#include "ToyPtr.h"

int main(int argc, char **argv) {
  // We want two pointers!
  ToyPtr<int> x(new int(5));
  ToyPtr<int> y = x;
  return 0;
}
```



!! Double delete !!

What Makes This a Toy?

Can’t handle:

- Arrays
  - Needs to use `delete[]`
- Copying
- Reassignment
- Comparison
- … plus many other subtleties…

Luckily, others have built non-toy smart pointers for us!

unique_ptr

Smart Pointers pro

std::unique_ptr

A `unique_ptr` is the ***sole owner*** of a pointer

- A template: template parameter is the type that the "owned" pointer references (i.e., the `T` in pointer type `T*`)
- Part of C++'s standard library (C++11)
- Once we give a vanilla pointer a unique_ptr, we should stop using the original (non-smart) pointer
- Its destructor invokes `delete` on the owned pointer
  - Invoked when `unique_ptr` object is `delete`'d or falls out of scope via the `unique_ptr` destructor

Guarantees uniqueness by disabling copy and assignment.

`std::unique_ptr`

A `unique_ptr` is the ***sole owner*** of a pointer

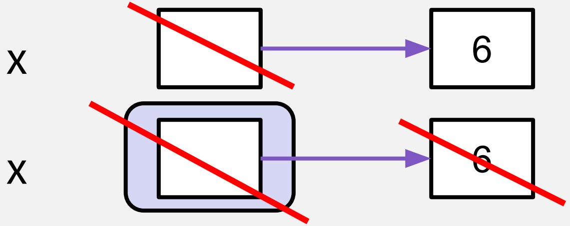
- A template: template parameter is the type that the "owned" pointer references (i.e., the `T` in pointer type `T*`)
- Part of C++'s standard library (C++11)
- Once we give a vanilla pointer a unique_ptr, we should stop using the original (non-smart) pointer
- Its destructor invokes `delete` on the owned pointer
  - Invoked when `unique_ptr` object is `delete`'d or falls out of scope via the `unique_ptr` destructor

Guarantees uniqueness by disabling copy and assignment.

Using `unique_ptr`

unique.cc

```
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::unique_ptr
#include <cstdlib>    // for EXIT_SUCCESS

void Leaky() {
  int* x = new int(5);  // heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, therefore leak

void NotLeaky() {
  std::unique_ptr<int> x(new int(5));  // wrapped, heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, but no leak

int main(int argc, char **argv) {
  Leaky();
  NotLeaky();
  return EXIT_SUCCESS;
}
```



Why are `unique_ptr`s useful?

If you have many potential exits out of a function, it's easy to forget to call `delete` on all of them

- `unique_ptr` will `delete` its pointer when it falls out of scope
- Thus, a `unique_ptr` also helps with *exception safety*

```
void NotLeaky() {
  std::unique_ptr<int> x(new int(5));
  ...
  // lots of code, including several returns
  // lots of code, including potential exception throws
  ...
}
```

`unique_ptr`s Cannot Be Copied

`std::unique_ptr` has disabled its copy constructor and assignment operator

- You cannot copy a `unique_ptr`, helping maintain "uniqueness" or "ownership"

uniquefail.cc

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

int main(int argc, char **argv) {
  std::unique_ptr<int> x(new int(5));  // OK

  std::unique_ptr<int> y(x);           // fail – no copy ctor

  std::unique_ptr<int> z;              // OK – z is nullptr

  z = x;                               // fail – no assignment op

  return EXIT_SUCCESS;
}
```

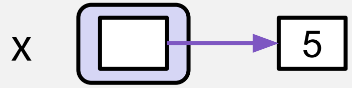
Transferring Ownership

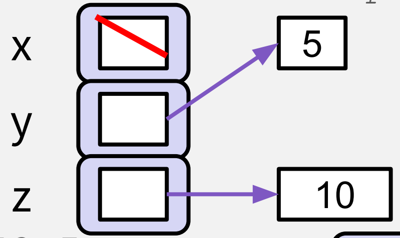
Use **reset**() and **release**() to transfer ownership

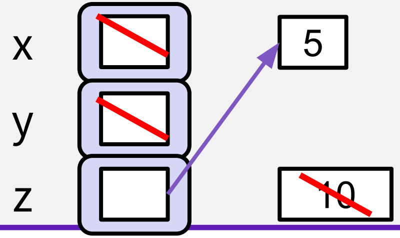
- **release** returns the pointer, sets wrapped pointer to `nullptr`
- **reset** `delete`'s the current pointer and stores a new one

uniquepass.cc

```
int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y(x.release());  // x abdicates ownership to y
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z.reset(y.release());
  return EXIT_SUCCESS;
}
```







unique_ptr and Arrays

unique_ptr can store arrays as well

- Will call delete[] on destruction

uniquearray.cc

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

using namespace std;

int main(int argc, char **argv) {
  unique_ptr<int[]> x(new int[5]);

  x[0] = 1;
  x[2] = 2;

  return EXIT_SUCCESS;
}
```

Demo: `unique_ptr` and Array

Questions?

shared_ptr
weak_ptr

Reference counting and more smart pointers…

What is Reference Counting?

Idea: associate a *reference count* with each object

- Reference count holds number of references (pointers) to the object
- Adjusted whenever pointers are changed:
  - Increase by 1 each time we have a new pointer to an object
  - Decrease by 1 each time a pointer to an object is removed
- When reference counter decreased to 0, no more pointers to the object, so delete it (automatically)

Used by C++ shared_ptr, not used in general for C++ memory management

Example

Suppose for the moment that we have a new C++ -like language that uses reference counting for heap data

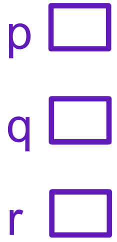
As in C++, a struct is a type with public fields, so we can implement lists of integers using the following Node type

```
struct Node {
  int payload;   // node payload
  Node* next;   // next Node or nullptr
};
```

The reference counts would be handled behind the scenes by the memory manager code – they are not accessible to the programmer

Example 1

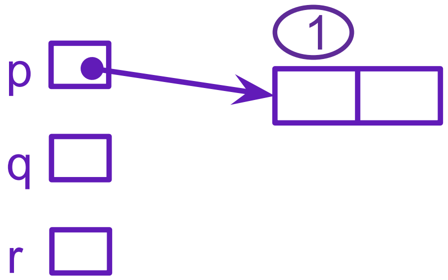
Let’s execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Example 1

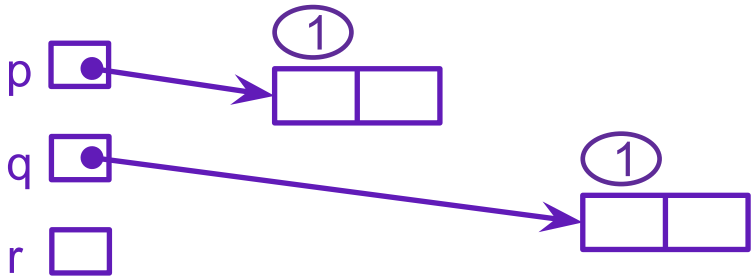
Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Example 1

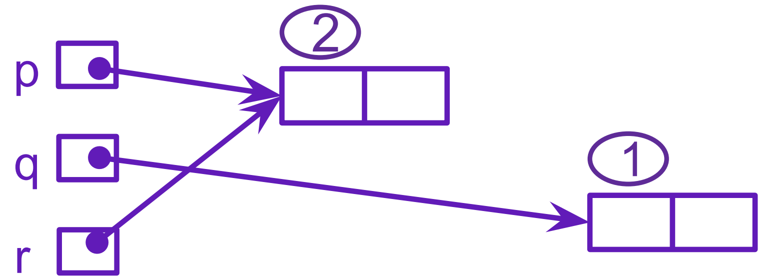
Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Example 1

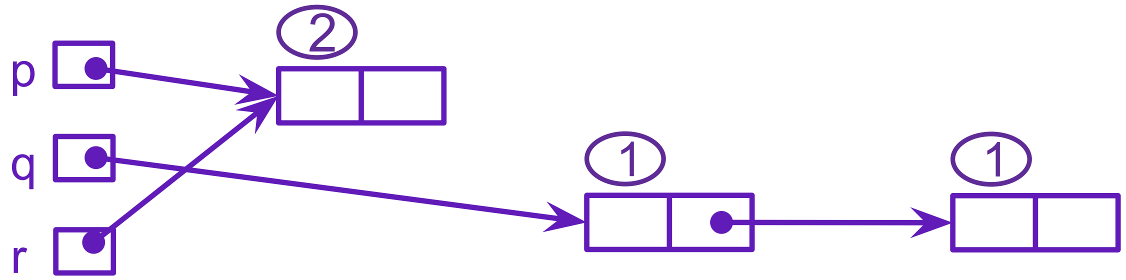
Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Example 1

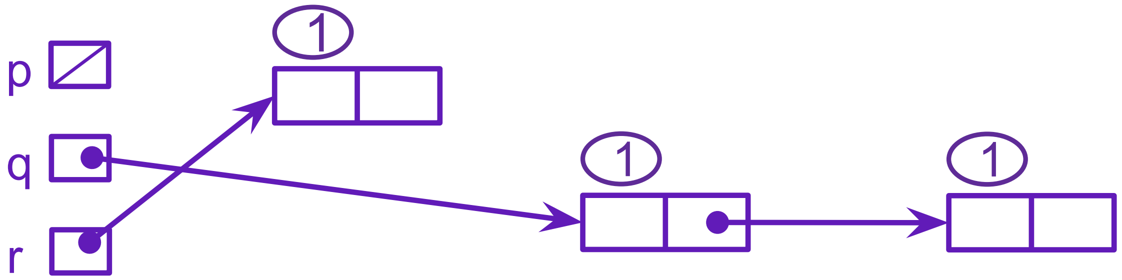
Let’s execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Example 1

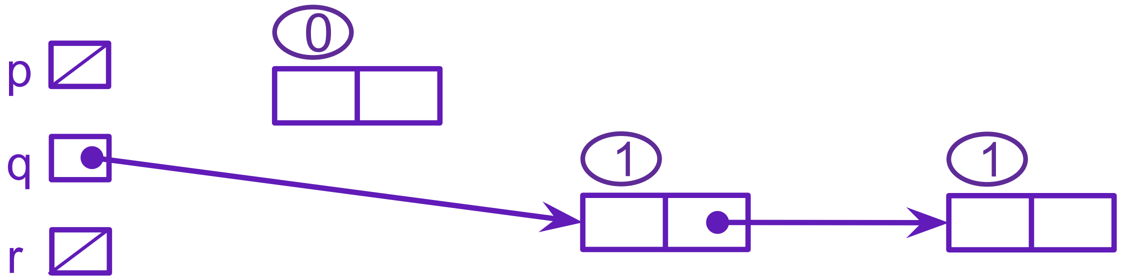
Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Example 1

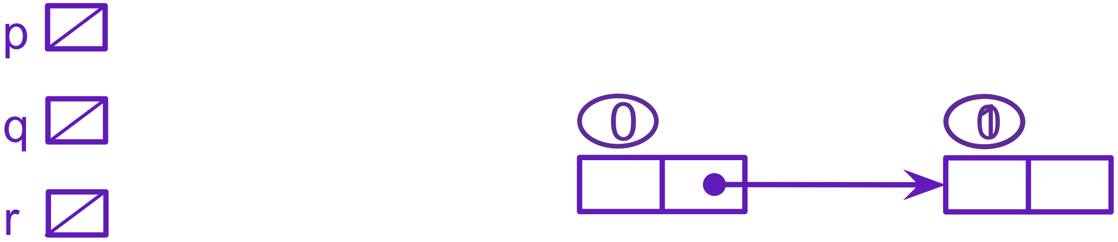
Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Example 1

Let’s execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

`std::shared_ptr`

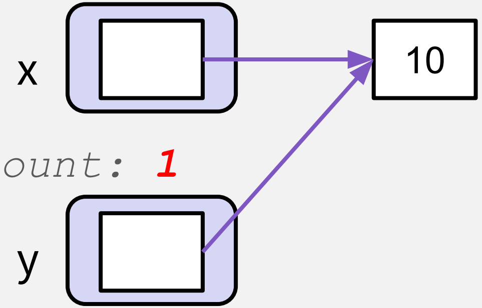
`shared_ptr` is similar to `unique_ptr` but we allow shared objects to have multiple owners

- The copy/assign operators are not disabled and *increment* reference counts as needed
  - After a copy/assign, the two `shared_ptr` objects point to the same pointed-to object and the (shared) reference count is incremented by **1**
- When a `shared_ptr` is destroyed, the reference count is *decremented*
  - When the reference count hits **0**, we `delete` the pointed-to object!
- Allows us to create complex linked structures (double-linked lists, graphs, etc.) at the cost of maintaining reference counts

shared_ptr Example

shared.cc

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr

int main(int argc, char **argv) {
  std::shared_ptr<int> x(new int(10));  // ref count: 1

  // temporary inner scope with local y (!)
  {
    std::shared_ptr<int> y = x;         // ref count: 2
    std::cout << *y << std::endl;
  }                                     // exit scope, y deleted

  std::cout << *x << std::endl;         // ref count: 1

  return EXIT_SUCCESS;
}                                       // ref count: 0
```



`shared_ptr`s and STL Containers

Safe to store `shared_ptr`s in containers, since copy & assign maintain a shared reference count; Also avoid extra object copies

sharedvec.cc

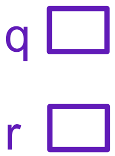
```
vector<std::shared_ptr<int>> vec;

vec.push_back(std::shared_ptr<int>(new int(9)));
vec.push_back(std::shared_ptr<int>(new int(5)));
vec.push_back(std::shared_ptr<int>(new int(7)));

int& z = *vec[1];
std::cout << "z is: " << z << std::endl;

std::shared_ptr<int> copied = vec[1];  // works!
std::cout << "*copied: " << *copied << std::endl;

vec.pop_back();  // removes smart ptr & deallocate 7
```

Demo: `shared_ptr` and STL

Questions?

Example 2

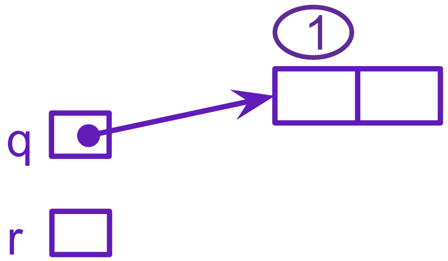
Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

Example 2

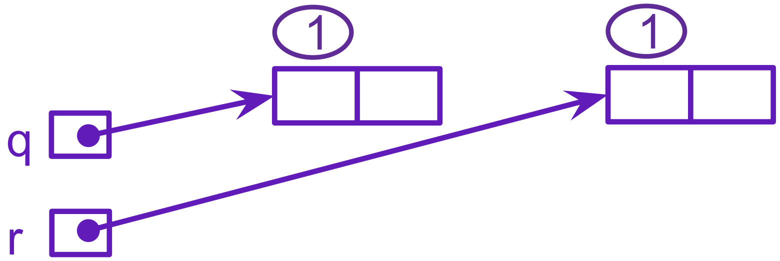
Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

Example 2

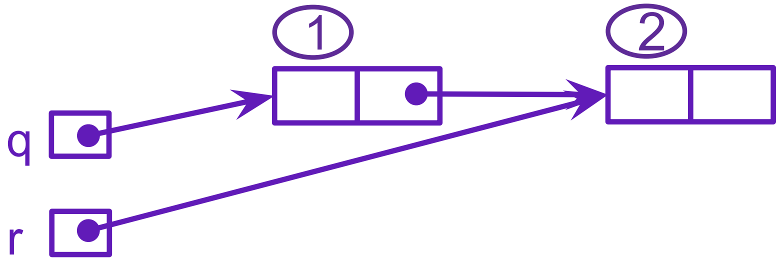
Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

Example 2

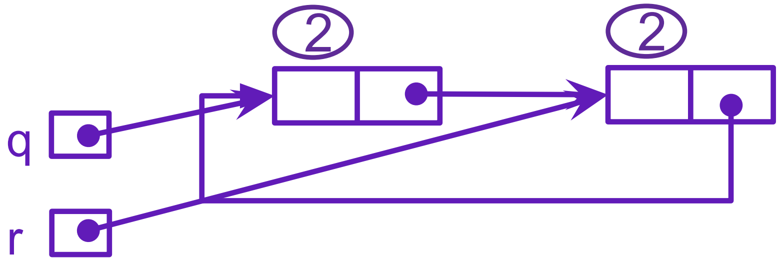
Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

Example 2

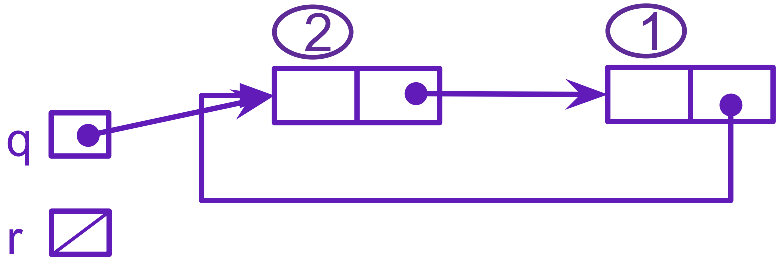
Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

Example 2

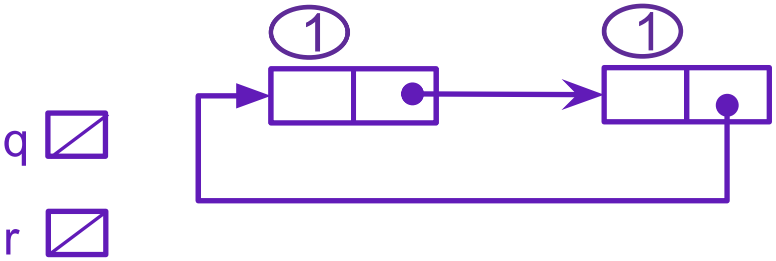
Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

Example 2

Similar to the previous code, but slightly different



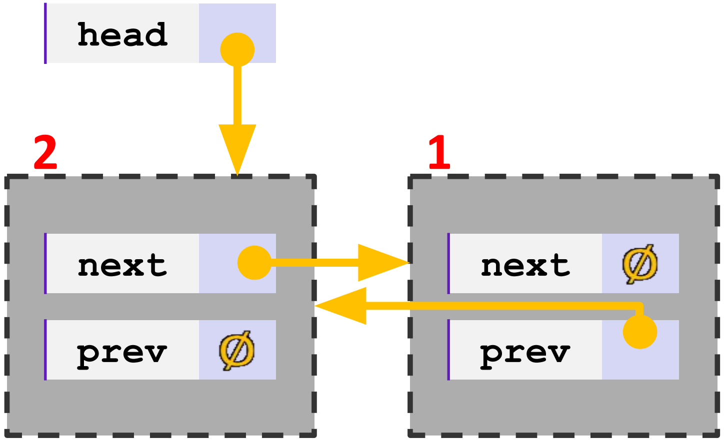
Memory leak!

```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

Cycle of `shared_ptrs`

sharedcycle.cc

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  shared_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```

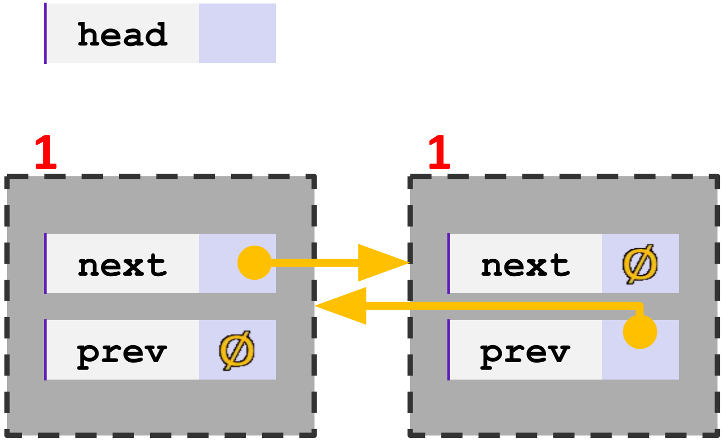


What happens when we `delete head`?

Cycle of `shared_ptrs`

sharedcycle.cc

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  shared_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```



What happens when we `delete head`? Nodes unreachable but not deleted because ref counts > 0

`std::weak_ptr`

`weak_ptr` is similar to a `shared_ptr` but doesn't affect the reference count

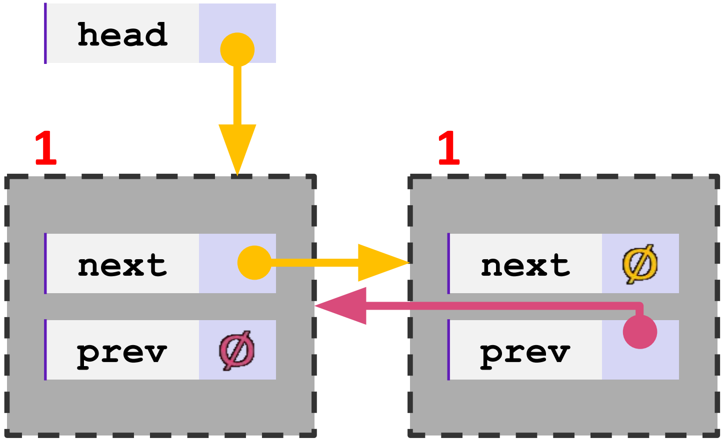
- Can *only* "point to" an object that is managed by a `shared_ptr`
- Not *really* a pointer – can't actually dereference unless you "get" its associated `shared_ptr`
- Because it doesn't influence the reference count, `weak_ptr`s can become "*dangling*"
  - Object referenced may have been `delete`'d
  - But you can check to see if the object still exists

Can be used to break our cycle problem!

Breaking the Cycle with weak_ptr

weakcycle.cc

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;
  return EXIT_SUCCESS;
}
```

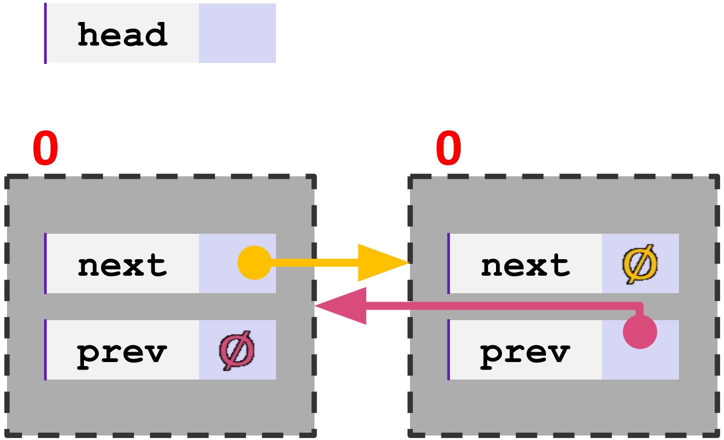


Now what happens when we `delete` head?

Breaking the Cycle with `weak_ptr`

weakcycle.cc

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;
  return EXIT_SUCCESS;
}
```

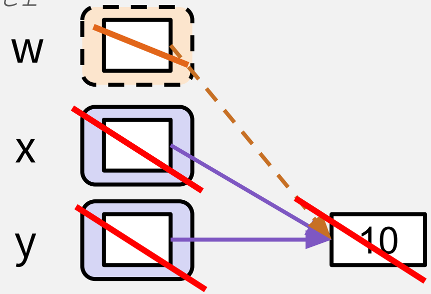


Now what happens when we `delete head`? Ref counts go to 0 and nodes deleted!

Using a weak_ptr

usingweak.cc

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr, std::weak_ptr
int main(int argc, char **argv) {
  std::weak_ptr<int> w;
  {  // temporary inner scope with local x
    std::shared_ptr<int> x;
    {  // temporary inner-inner scope with local y
      std::shared_ptr<int> y(new int(10));
      w = y;          // weak ref; ref count for "10" node is same
      x = w.lock();   // get "promoted" shared_ptr, ref cnt = 2
      std::cout << *x << std::endl;
    } // y deleted; ref count now 1
    std::cout << *x << std::endl;
  }                   // x deleted; ref count now 0; mem freed
  std::shared_ptr<int> a = w.lock();   // nullptr
  std::cout << a << std::endl;         // output is 0 (null)
  return EXIT_SUCCESS;
}
```



Demo: `weak_ptr` fixed code

Lecture Summary

A `unique_ptr` ***takes ownership*** of a pointer

- Cannot be copied, but can be moved
- Use **`release`**`()` to release ownership and stop managing the pointer for you
- **`reset`**`()` `delete`s old pointer value and stores a new one

A `shared_ptr` allows shared objects to have multiple owners by doing *reference counting*

- `delete`s an object once its reference count reaches zero

A `weak_ptr` works with a shared object but doesn't affect the reference count

- Can't actually be dereferenced, but can check if the object still exists and can get a `shared_ptr` from the `weak_ptr` if it does

Some Important Smart Pointer Functions

std::unique_ptr U;

- U.get() — Returns the raw pointer U is managing (⚠️ **Dangerous!**)
- U.release() — U stops managing its raw pointer and returns the raw pointer
- U.reset(q) — U cleans up its raw pointer and takes ownership of q

std::shared_ptr S;

- make_shared<T>(args) — Returns a shared_ptr pointer of a heap-allocated object
  shared_ptr<int> p3 = make_shared<int>(42);
- S.use_count() — Returns the reference count
- S.unique() — Returns true iff S.use_count() == 1

std::weak_ptr W;

- W.lock() — Constructs a shared pointer based off of W and returns it
- W.use_count() — Returns the reference count
- W.expired() — Returns true iff W is expired (W.use_count() == 0)

Questions?

Caution

Smart pointers are smart… ?🧐

“Smart” Pointers

Smart pointers still don’t know everything, you must be careful with what pointers you give it to manage.

- Smart pointers can’t tell if a pointer is on the heap or not.
- Still uses `delete` on default.
- Smart pointers can’t tell if you are re-using a raw pointer.
- Don’t point smart pointers at the stack.

Using a non-heap pointer

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

int main(int argc, char **argv) {
    int x = 374;
    shared_ptr<int> p1(&x);
    return EXIT_SUCCESS;
}
```

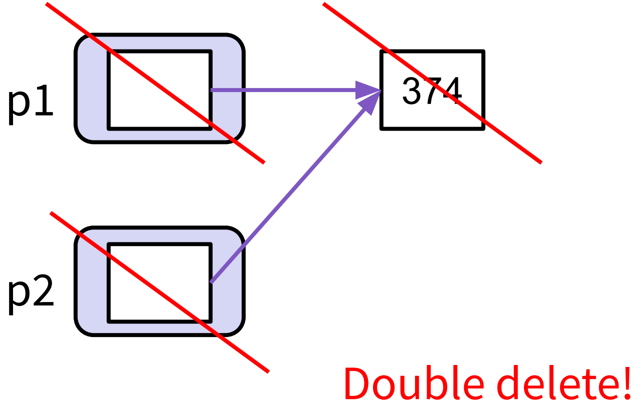
Smart pointers can't tell if the pointer you gave points to the heap!

- Will still call delete on the pointer when destructed.

Re-using a raw pointer

```
#include <cstdlib>
#include <memory>

using std::unique_ptr;

int main(int argc, char **argv) {
    int* x = new int(374);

    unique_ptr<int> p1(x);

    unique_ptr<int> p2(x);

    return EXIT_SUCCESS;
}
```

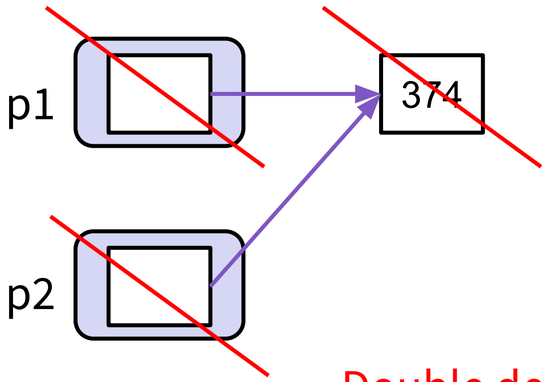
Smart pointers can’t tell if you are re-using a raw pointer.



Re-using a raw pointer

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

int main(int argc, char **argv) {
    int* x = new int(374);

    shared_ptr<int> p1(x);

    shared_ptr<int> p2(x);

    return EXIT_SUCCESS;
}
```

Smart pointers can't tell if you are re-using a raw pointer.



Double delete!

Automatic memory management

Different paradigms

	Tracing (mark & sweep)	Reference counting
Method	Mark all variables reachable from root objects, then sweep remaining ones	Automatically frees memory when `ref_count == 0`
Language	Java	C++ w/ Smart Pointers
Perf cost	Running the garbage collector can pause the entire program	Added overhead to every allocation/deallocation and assignment
Possible issues	Dangling references, GC behavior might be unpredictable	Cycles, overhead